

# The Level Ancestor Problem in Practice

Dimitris Papamichail<sup>1</sup>, Thomas Caputi<sup>1</sup>, and Georgios Papamichail<sup>2</sup>

<sup>1</sup> Department of Computer Science,  
The College of New Jersey,  
Ewing, NJ, USA  
papamicd@tcnj.edu  
www.tcnj.edu/~papamicd

<sup>2</sup> Department of Informatics,  
New York College,  
Athens, Greece

**Abstract.** Given a rooted tree  $T$ , the level ancestor problem is the problem of answering queries of the form  $LA(v, d)$ , which identify the level  $d$  ancestor of a node  $v$  in the tree. Several algorithms of varied complexity have been proposed in the literature, including optimal solutions that preprocess  $T$  in linear bounded time and proceed to answer queries in constant bounded time. Despite its significance and numerous applications, to date there have been no comparative studies of the performance of these algorithms and no implementations are widely available. In our experimental study we have implemented and compared several solutions for the level ancestor problem, including two optimal algorithms, and examine their space requirements and time performance.

## 1 Introduction and background

The Level Ancestor (LA) problem is a fundamental problem on trees, which is defined as follows: Given a rooted tree  $T$  with  $n$  nodes, and queries of the form  $LA(v, d)$ , where  $v$  is a tree node and  $d$  an integer, find the depth  $d$  ancestor of node  $v$ , meaning the  $d$ th vertex on the path from the root to  $v$ . Naively, such queries can be answer in  $O(n)$  without the use of any auxiliary data structure, or alternatively one could precompute all possible queries using  $O(n^2)$  time and space, allowing for constant time bounded queries.

The level ancestor problem is related to the extensively studied Least Common Ancestor (LCS) problem. Indeed, Harel and Tarjan [11] use level ancestors as a subroutine of an LCA algorithm. Level ancestors are used as part of space-efficient ordinal trees [10], which can be used for the representation of XML documents that support XPath queries. Level ancestor queries are part of the primitive operations that are supported by other compressed data structures [13] and are used in range-aggregate queries in trees [14].

There are several known optimal solutions to the level ancestor problem with  $O(n)$  complexity for preprocessing and  $O(1)$  query time. Dietz [8] first published an optimal algorithm for the dynamic version of the problem, where Berkman

and Vishkin [5] published an optimal parallel (PRAM) algorithm, albeit involving unwieldy constants of the order of  $2^{28}$ . It is noteworthy that complex algorithms for the dynamic and parallel versions of the level ancestor problem preceded simpler algorithms for the static serial version. A substantially simplified algorithm for the dynamic and static variants was published by Alstrup and Holm [2], and an even simpler optimal algorithm – due to its progressive construction in stages – for the static version was given by Bender and Farach-Colton [4]. Ben-Amram [3] contributed yet another simple algorithm for the static level ancestor problem in a technical report, along with an efficient implementation.

The remainder of the paper is organized as follows. In Section 2 we provide definitions and review the algorithms involved in this study. In Section 3 we describe the experiments performed and the results obtained. In Section 4, we discuss the results and draw conclusions about the advantages and limitations of the different algorithms compared.

## 2 Definitions, experimental design and implementations

Following the notation and definitions in [4], the *depth* of a node  $v$  in tree  $T$ , denoted as  $depth(v)$ , is the number of edges on the shortest path from  $v$  to the root. The root itself has depth 0. The *height* of a node  $v$  in tree  $T$ , denoted  $height(v)$ , is the number of nodes on the path from  $v$  to its deepest descendant. The leaves have height 1. The level ancestor of node  $v$  at depth  $d$ , denoted as  $LA(v, d)$ , is a node  $u$ , such that  $u$  is an ancestor of  $v$  and  $depth(u) = d$ . If such a node does not exist, then  $LA(v, d)$  is undefined. The algorithms described in this paper have both preprocessing and query time complexity. An algorithm that has preprocessing time  $f(n)$  and query time  $g(n)$  will be denoted as having complexity  $\langle f(n), g(n) \rangle$ .

For the purpose of this study, we have implemented five algorithms for the level ancestor problem, each with a distinct time and space complexity. Four of algorithms are components of the optimal algorithm of Bender and Farach-Colton, described in [4], and are eventually combined. The five algorithms implemented are listed below, together with their preprocessing and query time bounds:

1. Table algorithm –  $\langle O(n^2), O(1) \rangle$
2. Jump-Pointers algorithm –  $\langle O(n \log n), O(\log n) \rangle$
3. Ladder algorithm –  $\langle O(n), O(\log n) \rangle$
4. Jump-Ladder algorithm –  $\langle O(n \log n), O(1) \rangle$
5. Macro-Micro-Tree algorithm –  $\langle O(n), O(1) \rangle$

Each of these algorithms have their own efficiency/simplicity trends. In addition to these five algorithm implementations, our performance evaluations include another optimal algorithm by Ben-Amram, described in [3].

## 3 Level Ancestor Implementations

For all the implementations, the tree is stored in an array, based on the *Depth First Search* (DFS) numbering. This representation, which is equivalent of having

a tree style structure for the static Level Ancestor problem, helps avoid a level of indirection when locating a node based on its DFS number. In addition, a DFS procedure only has to scan the table serially. This implementation can be extended for dynamic approaches, albeit with loss of some of the advantages mentioned. The space occupied by this representation is optimal. Every node stores its parent, left and right child pointers, as well as a field for the depth of the node, which is needed in all algorithms we will encounter (except the table algorithm).

### 3.1 The Table Algorithm

The table algorithm for the level ancestor problem is the naive solution that preprocesses and stores all  $O(n^2)$  queries. The simplest of all algorithms presented, it is implemented with dynamic programming, given the Euler representation of the tree, which we will call the *tree signature*. A 2-dimensional table holds the level ancestors of all nodes. The ancestor list of each node can be progressively generated from the parent node ancestor list, when traversing the tree in the DFS order.

This is an optimal query time algorithm, able to answer level ancestor queries by performing only two memory references per query, an attribute which makes this algorithm superior to any other solution when space is not an issue. The Table algorithm consists a compelling solution when dealing with balanced trees or at least trees with logarithmic expected node depth, and the programming language of choice supports "ragged" arrays (or arrays of arrays), as is the case with C and Java. Preprocessing time is roughly proportional to the space used to store the array.

### 3.2 The Jump-Pointer Algorithm

The jump-pointer algorithm associates every node in the tree with a list of pointers to its ancestor nodes, that allow to "jump" up the tree by powers of 2. Specifically, for every node  $v$  in the tree, there exists a list associated with  $v$  that contains pointers to all  $2^i$ th ancestors of  $v$ , for  $0 \leq i \leq 2^{\lceil \log \text{depth}(v) \rceil}$ .

Using dynamic programming, the pointer lists can be generated in  $O(n \log n)$  time by an Euler traversal of the tree. Queries can be processed in  $O(\log n)$  time by following jump pointers up the tree, covering at least half of the remaining distance to the desired ancestor with each jump.

### 3.3 The Ladder Algorithm

This algorithm starts with a longest path decomposition of the tree  $T$ , into non-disjoint paths called *ladders*. It proceeds by extending the ladders towards the root, up to twice their original size. This latter action creates the property that every node of *height*  $h$  belongs to a ladder that includes its ancestor of height at least  $2h$ , or the root of the tree, allowing for queries in  $O(\log n)$  time.

This algorithm can be implemented by a bottom-up scan of the tree, in which the heights of the nodes are calculated. Ladders can be constructed recursively, starting from the deepest node and queuing nodes encountered while traversing the current ladder, which are subsequently processed. Careful implementation leads to a linear time and space algorithm, which, as will be shown in the experimental section below, leads to significant space savings, while still being competitive in the time domain.

### 3.4 The Jump-Ladder Algorithm

The jump-ladder algorithm combines the preprocessing benefits of the jump-pointer and ladder algorithms to achieve constant time queries. Since the jump-pointer algorithm makes exponentially decreasing hops up the tree and the ladder algorithm makes exponentially increasing hops up the tree, following a single jump pointer and then climbing a single ladder leads to the desired level ancestor in two steps.

The preprocessing stage combines the jump-pointer and ladder algorithm preprocessing procedures, resulting in  $O(n \log n)$  complexity in time and space, where the actual space used is the sum of the space required by both component algorithms.

### 3.5 The Macro-Micro-Tree Algorithm

The Macro-Micro-Tree algorithm combines and extends the jump-pointer and ladder algorithms. Since ladders can be preprocessed in linear time, they are used without modifications. Jump pointers though are not assigned to all nodes – since the preprocessing would then require  $O(n \log n)$  space and time – but only to a specific set of nodes, which are called *jump nodes*. Ancestors of jump nodes can use their jump pointers, and are called *macro nodes*, forming a connected subtree of  $T$  called the *macrotree*. Descendants of jump nodes form disconnected *microtrees*. By applying the standard data structural technique in [9] and limiting the size of the microtrees to  $\lceil \log n/4 \rceil$ , the number of jump nodes is bounded by  $n/\log n$  and the total number of jump pointers becomes  $O(n)$ . Due to their limited size, it has been shown that microtrees adopt a limited number of shapes,  $O(\sqrt{n})$ , which can be precomputed and stored using the table algorithm, also in linear bounded time.

The Macro-Micro algorithm has optimal time complexity, but also significant implementation complexity, with quite a few details and subtleties on top of the other four implementations. In addition to the node depth and height precalculations, the *weight* of each node, as in the size of the subtree rooted at a node, has to be calculated in order to characterize Micro and Macro nodes. Overall there are four distinct types of nodes, the Micro, the Macro, the Micro-Root (roots of the Micro trees) and the Jump nodes, the latter been associated with jump pointers, which can be utilized by their Macro ancestors. Extra space is required to store auxiliary pointers for each node, in addition to data structures that hold the tables for the Micro trees.

To locate appropriate entries in the tables of the microtrees, the microtree signatures are enumerated and stored, being indexed when probed by level ancestor queries. The queries themselves have multi-case functionality, using a different approach to locate the level ancestor of a node based on its type.

### 3.6 Ben-Amram’s Find-Smaller algorithm

Another relatively simple optimal algorithm for the static level ancestor problem was more recently presented by Ben-Amram [3], which is also based on the microset technique [11,9]. This algorithm uses the Euler tour representation of a rooted tree and is a simplification of Berkman and Vishkin’s PRAM algorithm [5]. It reduces the level ancestor problem to the *Find-Smaller* problem, the goal of which is, given an array  $A$ , an index  $v$  and an integer  $d$ , to find the minimal  $u > v$  such that  $A_u \leq d$ . Auxiliary array data structures are created to support efficient queries and the microset technique is used to sparsify these data structures.

Ben Amram’s algorithm was implemented in C by Victor Buchnik and can be obtained from the authors. We used the available implementation to compare with the algorithms implementations that were described in the previous subsections. We will refer to this algorithm as the *Find-Smaller* algorithm.

### 3.7 Tree representation and random generation

All level ancestor algorithms work on trees, which consist part of the input of the algorithm. To represent rooted trees, we decided to use the Euler representation, a unique mapping of a tree to an array which is compact and is derived by an Euler traversal of the tree, commonly referred to as *depth first search* (DFS). For our implementations, we selected to use a binary representation of the Euler traversal, where ‘1’ represents forward downward traversal of edges towards descendant nodes, and ‘0’ represents upward traversal of edges toward ancestor nodes. A rooted tree with  $n$  nodes can therefore be represented by  $2n - 2$  binary digits, which we call the *signature* of the tree. It should be noted that, for ordinal binary trees of large size, this is the most compact representation. For the purpose of our study, we have used an ASCII representation of the binary digits of the Euler traversal, to facilitate visual inspection and debugging. Given a rooted tree signature, it is trivial to construct a pointer linked or table based tree structure in linear time on the size of the tree.

To generate random trees for our experiments, we use the split subtree method [7], where, in order to create a tree with  $n$  nodes, we generate a real-valued random variable  $x$  in the unit interval  $(0, 1)$ , assign  $\lfloor xn \rfloor$  nodes to the left subtree, one node to the root, and the rest to the right subtree. The process continues recursively for each subtree. This method generates trees equivalent to the ones generated with the random binary search tree process by using random permutations [12], where any node has an equal probability to be chosen as root.

This method has several advantages when constructing random trees. It can be used in a straightforward manner to construct the Euler traversal sequence (signature) of a random tree with  $n$  nodes. First we allocate a table with  $2n - 2$

positions and select a random number  $k = \lfloor xn \rfloor$  between 1 and  $n$ . We can then insert ‘1’ in positions 0 and  $2k$ , ‘0’ in positions  $2k - 1$  and  $2n - 3$ , and recursively process the two subtrees and corresponding subarrays (or single subtree/subarray when  $k = n$ ). The expected average depth of a node of a random tree created with the split subtree method is  $O(\log n)$  [1], while the expected depth of the tree is also bounded logarithmically [6]. These observations have been validated in the experiments described in the next section. The split subtree method also is easily modifiable to accommodate skewed unbalanced trees, by varying the range of values that the random variable takes. Using that property we have generated skewed trees with higher average node depth, as described in the next section.

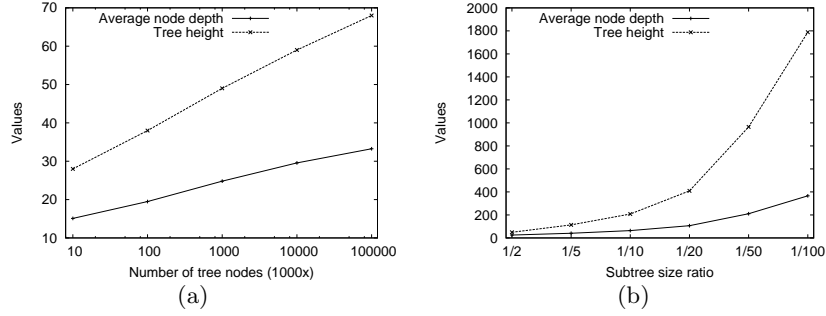
## 4 Algorithm Evaluation

The first five algorithms described in the previous section were implemented in the programming language C, using standard libraries. All programs were compiled with gcc version 4.8.1, on the Ubuntu 13.10 (saucy) operating system, running on an Intel Haswell i7-4900MQ CPU (22nm lithography) at 2.8GHz, on a single core and thread. Each core has 64KB of L1 cache, 256KB of L2 cache, and shares 8MB of L3 cache with the other cores. The machine used has 32GB of main memory and test cases have been limited in size in order for program instructions and data to reside in main memory. For this study we do not examine disk paging, since several of the algorithms vary significantly in their space usage, which we expect to be a predominant factor in performance when using virtual memory. All programs were compiled with the ‘-O3’ optimization level set.

Randomly generated trees were varied in size between 100K and 204.8M nodes, with sizes distributed evenly logarithmically in doubled size steps. For each size step, 10 random trees were generated and the time and space usage results were averaged between these 10 cases. The number of random level ancestor queries generated and run for each experiment were kept constant at 100M, which we deemed sufficient to run experiments long enough not to be influenced by process swapping, interrupts and other operating system events. We decided not to vary the number of queries for each experiment, since query execution commences after static structures are in place following their preprocessing, and we expect the execution time to be a linear function of the number of queries. Tree depth and average node depth for the random trees generated are depicted in Figure 1(a).

Applications for the level ancestor problems quite often involve trees that are not balanced and may have properties, such as depth, that are not well represented by the expected evenly distributed random trees described above. To examine the behavior of the algorithms when run on unbalanced trees, we created a set of skewed rooted trees with 1M nodes each, varying the size ratio of the subtrees at each node. Using the same split subtree method, we limited the

range of the random variable to a fraction of the unit interval. The properties of these skewed trees generated for varying ratios are shown in Figure 1(b).



**Fig. 1.** Tree properties. (a) Randomly generated split trees. (b) Randomly generated split trees with bounded subtree size ratio.

It should be noted that the Find-Smaller algorithm uses a different format for the input trees, which are also represented by their Euler traversals, but each node has a unique number that is used to store its position. As such, input trees for the Find-Smaller implementation occupy significantly more space on disk, but this does not affect any of our performance evaluations, since we only measure space utilization in resident memory, while time measurements commence only after trees are fully loaded into memory.

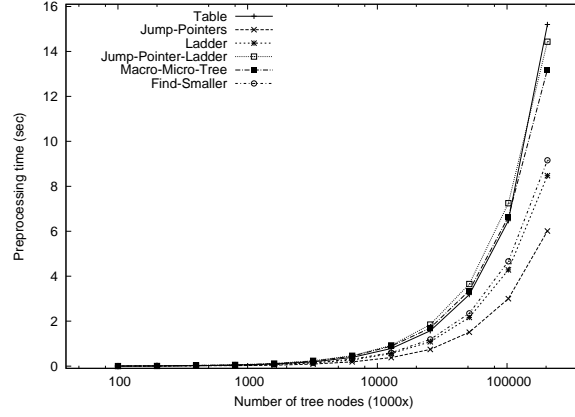
#### 4.1 Preprocessing

Preprocessing involves the time required to build necessary auxiliary data structures that support each algorithm, and excludes the time it takes to load the tree from a file. Experimental preprocessing times for all six algorithms and varying tree sizes, for the randomly generated trees with expected logarithmic height, are shown in Figure 2.

All algorithms can preprocess trees of size up to 205M nodes in less than 16 seconds, with the Jump-Pointer algorithm outperforming the rest with less than 6 seconds preprocessing for 205M nodes. Since preprocessing only occurs once for static trees, no algorithm has a clear disadvantage. In the case of skewed trees, all algorithms except the Table algorithm perform equally well (data not shown). The Table algorithm experiences preprocessing time increases proportional to its space usage, as analyzed below.

#### 4.2 Space Requirements

Space is probably the most critical factor in selecting the appropriate level ancestor algorithm. Queries are performed in constant time and preprocessing is



**Fig. 2.** Preprocessing times for evenly distributed randomly generated trees

expedient even for very large trees. Since any solution will be satisfactory when processing small trees in modern workstations, it is large trees and the extra storage space involved which will influence most significantly the selection of one solution over another. The space utilization of the six implementations after preprocessing is presented in Tables 1 and 2.

**Table 1.** Space usage (MB) of level ancestor algorithms for random trees with log bounded expected depth

Nodes (1000x)	Table	Jump- Pointer	Ladder	Jump- Ladder	Macro- Micro-Tree	Find- Smaller
100	13	7	6	10	8	6
200	27	13	11	19	15	12
400	56	26	22	38	30	23
800	114	51	44	75	59	44
1600	239	101	87	149	116	89
3200	492	201	172	297	232	169
6400	1020	401	344	594	463	338
12800	2119	801	687	1188	926	679
25600	4362	1601	1374	2374	1832	1361
51200	8966	3201	2748	4748	3664	2625
102400	18578	6401	5496	9496	7327	5255
204800	36700	12801	10992	18992	14654	10568

The Find-Smaller algorithm is a clear choice for space efficiency, since, despite being worst case query-time optimal, it utilizes less space even than the suboptimal algorithms with simpler auxiliary data structures. It is followed by the Ladder algorithm, one of the simplest solutions for this problem.



**Table 2.** Space usage (MB) of level ancestor algorithms using skewed random trees

Ratio	Table	Jump-Pointer	Ladder	Jump-Ladder	Macro-Micro-Tree	Find-Smaller
1/2	147	63	52	91	71	56
1/5	208	64	50	90	71	56
1/10	298	70	49	95	70	56
1/20	466	76	48	100	69	57
1/50	871	78	46	100	64	58
1/100	1480	79	47	101	66	58

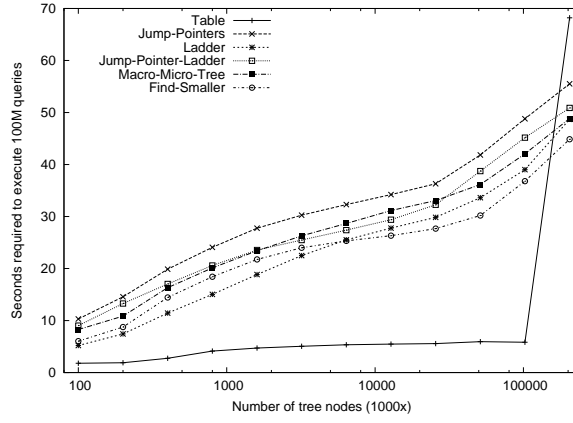
All algorithms have a linear space utilization dependence on the size of the tree, as demonstrated in the results of Table 1, except for the Table algorithm, which uses  $O(n \log h)$  space, with  $h$  being the average node depth of the tree. The last test case set with trees of 204.8M nodes forced the Table algorithm to use approximately 4GB of virtual memory, which generated a large number of page faults and severely affected the query performance, as observed in the next section.

In the case of unbalanced trees, as observed in Table 2, the Ladder and Find-Smaller algorithms continue to dominate other algorithms, but we notice an interesting trend in the space utilization of the Ladder algorithm, which seems to slightly improve with increased imbalance. This is not surprising, since unbalanced trees are expected to have fewer ladders as a result of the path decomposition, and in average fewer extra nodes when they are extended.

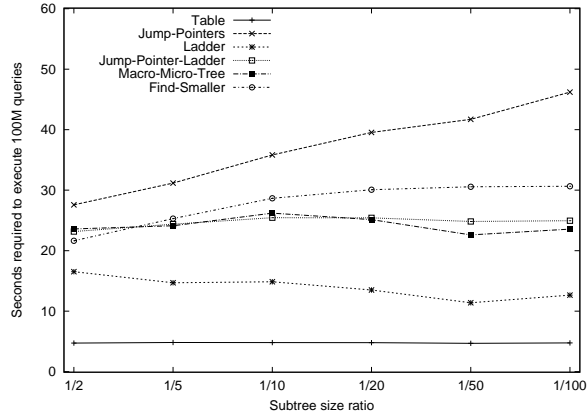
### 4.3 Query time performance

Level ancestor queries are often performed as subroutines in other algorithms. As such, it is critical that they are expediently executed. In figure 3 we can observe that the Table algorithm outperforms all others in query execution time, except when its increased space requirements force part of the data structures to be stored in virtual memory, as is the case with the last data point; this creates the distinct spike in the graph. All algorithms, including the query-optimal Table, Macro-Micro-Tree, Jump-Ladder, and Find-Smaller, experience increased query execution times as a function of tree size, which can be explained by increased cache misses, once the data structures do not fit in the different levels of cache of the processor. The Find-Smaller algorithm outperforms the rest (except for the Table algorithm) once tree sizes surpass 10M nodes.

When considering unbalanced trees, the Ladder algorithm performs surprising well, with query times that improve as imbalance increases. Once again, the expected fewer number of ladders as a result of the tree imbalance can lead to a reduced expected number of hops between ladders when executing the queries. The performance of all other algorithms seem to remain relatively stable with increased imbalance, except for the Jump-Pointer algorithm, which seems to re-



**Fig. 3.** Time to run 100M queries on randomly generated trees



**Fig. 4.** Time to run 100M queries on unbalanced randomly generated trees with 1M nodes. The x axis represents the maximum ratio between subtree sizes of node children.

quire an increasing number of jumps for nodes to reach their ancestors, as the average node depth of the tree increases.

## 5 Conclusions

Each algorithm presented, implemented and tested in this paper has distinct merits and shortcomings. No one algorithm has a clear advantage over the others in every aspect. The Table algorithm outperforms, as expected, all other solutions in query execution time, if one can afford its worst case quadratic space requirement, which can be significant even when the input tree has logarithmically bounded depth. Space utilization, which is possibly the most critical component influencing algorithm choice, seems to correlate with the simplicity of algorithms with data structures that use linear bounded space, with the Find-Smaller and the Ladder algorithms outperforming all others, the latter demonstrating increased benefits when dealing with unbalanced trees.

Some of the simpler non-optimal algorithms, such as the Ladder, lead to simple efficient implementations, that are easier to comprehend, maintain, optimize and extend. On the other hand, with ever increasing problem sizes and machine capabilities, the optimal algorithms, especially the Find-Smaller, will be the solution of choice once  $\log n$  becomes sizable. Ben-Amram's algorithm seems to be the most versatile solution, combining excellent query execution times with the smallest space footprint and reasonable preprocessing requirements.

The Table, Jump-Pointer, Ladder, Jump-Ladder and Macro-Micro-Tree algorithm implementations can be downloaded at [www.tcnj.edu/~papamicd/level\\_ancestor](http://www.tcnj.edu/~papamicd/level_ancestor). The Find-Smaller algorithm implementation can be obtained from its author [3].

## References

1. David Aldous. Probability distributions on cladograms. In *In Random Discrete Structures*, pages 1–18. Springer, 1996.
2. Stephen Alstrup and Jacob Holm. Improved algorithms for finding level ancestors in dynamic trees. In *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, number 1853 in LNCS*, pages 73–84. Springer Verlag, 2000.
3. Amir M. Ben-Amram. The euler path to static level-ancestors. *CoRR*, abs/0909.1030, 2009.
4. Michael A. Bender and Martín Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, June 2004.
5. Omer Berkman and Uzi Vishkin. Finding level-ancestors in trees. *Journal of Computer and System Sciences*, 48(2):214 – 230, 1994.
6. Luc Devroye. A note on the height of binary search trees. *J. ACM*, 33(3):489–498, May 1986.
7. Luc Devroye and Paul Kruszewski. The botanical beauty of random binary trees. In Franz Brandenburg, editor, *Graph Drawing*, volume 1027 of *Lecture Notes in Computer Science*, pages 166–177. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0021801.

8. Paul Dietz. Finding level-ancestors in dynamic trees. In Frank Dehne, Jrg-Rdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures*, volume 519 of *Lecture Notes in Computer Science*, pages 32–40. Springer Berlin / Heidelberg, 1991. 10.1007/BFb0028247.
9. Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, STOC '83, pages 246–251, New York, NY, USA, 1983. ACM.
10. Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. *ACM Trans. Algorithms*, 2(4):510–534, October 2006.
11. Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, May 1984.
12. C.C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, 2012.
13. Kunihiro Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, SODA '06, pages 1230–1239, New York, NY, USA, 2006. ACM.
14. Hao Yuan and Mikhail J. Atallah. Efficient data structures for range-aggregate queries on trees. In *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, pages 111–120, New York, NY, USA, 2009. ACM.